

## 30-程序库的设计：Moco是如何解决集成问题的？

你好，我是郑晔！

经过前面内容的讲解，我终于把软件设计的基础知识交付给你了，如果你有一定的经验，相信有很多东西你已经可以借鉴到日常工作中了。

但是对于一些同学来说，这些知识恐怕还是有些抽象。那在接下来的几讲中，我会给你讲几个例子，让你看看如何在日常的工作中，运用学到的这些知识，巩固一下前面所学。

我在[第9讲](#)说过，学习软件设计，可以从写程序库开始。所以，我们的巩固篇就从一个程序库开讲。这是我自己维护的一个开源项目 [Moco](#)，它曾经获得 2013 年的 Oracle Duke 选择奖。

Moco 是用来做模拟服务器的，你既可以把它当作一个程序库用在自动化测试里，也可以把它单独部署，做一个独立的服务器。我们先来看一个用 Moco 写的测试，感受一下它的简单吧！

```
public void should_return_expected_response() {
    // 设置模拟服务器的信息
    // 设置服务器访问的端口
    HttpServer server = httpServer(12306);
    // 访问/foo 这个 URI 时，返回 bar
    server.request(by(uri("/foo"))).response("bar");

    // 开始执行测试
    running(server, new Runnable() {
        // 这里用了 Apache HTTP库访问模拟服务器，实际上，可以使用你的真实项目
        Content content = Request.Get("http://localhost:12306/foo")
            .execute()
            .returnContent();

        // 对结果进行断言
        assertThat(content.asString(), is("bar"));
    });
}
```

这一讲，我就来说说它的设计过程，让你看看一个程序库是如何诞生以及成长的。

### 集成的问题

不知道你有没有发现，阻碍一个人写出一个程序库的，往往是第一步，也就是**要实现一个什么样的程序库**。因为对于很多人来说，能想到的程序库，别人都写了，再造一个轮子意义并不大。

但是，这种思路往往是站在理解结果的角度。其实，**程序库和所有的应用一样，都是从一个要解决的问题出发**。所以，在日常的繁忙工作中，我们需要偶尔抬头，想想哪些问题正困扰着我们，也许这就是一个程序库或者一个工具的出发点。

曾经有一个问题困扰了我好久，就是**集成**。还记得在我初入职场时，有一次，我们开发的系统要与第三方厂商的系统进行集成。可是，怎样才能知道我们与第三方集成的效果呢？我们想到的办法就是模拟一个第三方服务。

于是，作为当时的新人，我就承担起编写这个模拟服务的任务。那个时候还真是年少无知，居然自己写了一个 HTTP 服务器，然后又继续在上面写了应用协议。那时候的我完全没有编写程序库的意识，只是有人要求我返回什么样的应答，我就改代码，返回一个什么应答。

在我的职业生涯中，集成并不少见，只是后来我的经验多了，这种编写模拟服务的事就交到了别人的手上，我就成了那个让别人改来改去的人。

2012 年，我加入到一个海外合作的项目中，这个项目也有一个模拟的 HTTP 服务。开发人员根据自己的需要去改动代码，让这个模拟服务返回不同的应答。之后，他们再打出一个包，部署到一个 Web 服务器上。显然，这比我当年一个人维护模拟服务器要进步很多了，至少它不用考虑 HTTP 协议层面的问题了。

不过，依旧要自己部署模拟服务这一点，让我突然想起当年开发模拟服务时的景象。这么多年过去了，模拟服务却依然如此麻烦，没有得到任何好转，也许我可以做点什么。比起当年做软件开发的懵懂的我，工作了十多年的我，显然已经有了更多的知识储备。

## 从问题到需求，再到解决方案

那问题有了，我要怎么解决这个问题呢？我需要先把它变成一个可以下手解决的需求。首先，我要考虑的是，我希望这个模拟服务做成什么样子呢？

- 它可以支持配置，这样的话，我就不用每次都调整代码了；
- 它可以独立部署，因为部署到应用服务器上的方式实在不够轻量级；
- 它可以是一个通用的解决方案，因为我已经在多个不同的场景下遇到类似的问题。

除了这些正常的需求之外，我还有一个额外的小需求，就是希望它**有一个有表达性的 DSL**。因为我当时刚刚翻译完《领域特定语言》，特别想找个机会练练手。

以我当时的知识水平来看，配置肯定不是问题，这是任何一个程序员都可以做到的。独立部署，应该也可行，虽然当时还不流行嵌入式的 Web 服务器，但我还知道有 Netty 这样的网络编程框架，我稍微做了一点调研就发现，用它实现一个简单的 Web 服务器并不难。

问题就是，我怎样能把它做成一个通用的方案？

在设计中，其实最难的部分就在这里。一个特定的问题总有一个快速的解决方案，而要想**做成一个通用方案，它就必须是一个通用的模式。这就需要我们z把问题抽丝剥茧，把无关的信息都拿掉，才可能看到最核心的部分**。而进行这种分析的根基，同样是我们前面说过的分离关注点。

我找到的核心问题就是，模拟服务到底是做什么的呢？其实，它就是按照我预期返回相应的应答。对，一方面，我要表达出预期；另一方面，它要给出返回的结果。

当我想明白这一点之后，一段代码浮现在我的脑海中：

```
server.request("foo").response("bar");
```

对，这就是这个模拟服务器最简单的样子。当请求是“foo”的时候，它就给出对应的应答“bar”，这个结构非常适用于 HTTP 这种请求应答的结构。这段代码简直太合我的胃口了，因为它还是一段内部 DSL，声明出这个模拟服务器的行为，我的额外需求也得到了满足。

如果代码真的可以做成这个样子，那它应该就可以写在单元测试里了。和现在一比，动辄需要启动整个应用，做人工的集成测试，这简直是一个巨大的飞跃。而且，从开发效率上看，这简直就是数量级的提升。

不过，上面只是给出了设置服务器的样子，如果我们要把它写到单元测试里，还要考虑到如何去启动和关闭服务器。于是，一段单元测试的代码就浮现了出来：

```
public void should_return_expected_response() {
    HttpServer server = httpServer(12306);
    server.request("foo").response("bar");
    running(server, new Runnable() {
        Content content = Request.Post("http://localhost:12306")
            .bodyString("foo", ContentType.TEXT_PLAIN)
            .execute()
            .returnContent();
        assertThat(content.asString(), is("foo"));
    });
}
```

这就是 Moco 的第一个测试了。有了测试，我就该考虑如何让测试通过了。同时，测试帮我锁定了具体的目标，我还知道了可用的技术，剩下的就是把它实现出来了。

对于程序员而言，实现反而是最简单的。就这样，我花了一个周末的时间，翻着各种文档，让第一个测试通过了。如此一来，Moco 在实现上的技术难度就被突破了。

## 基础设计的诞生

接下来，我就要考虑 Moco 可以提供怎样的功能了。Moco 首先是一个 HTTP 的模拟服务器，所以，它需要对各种 HTTP 的元素进行支持。HTTP 的元素有哪些呢？其实，无非就是 HTTP 协议中可以看到 HTTP 协议版本、URI、HTTP 方法、HTTP 头和 HTTP 内容等等这些东西。

问题来了，如果我们要把 Moco 实现成一个通用的解决方案，我们就需要任意地组合这些元素，我们该如何设计呢？

你可能已经想到了，在前面我们讲函数式编程的组合性时，已经提到了要设计可以组合的接口。是的，Moco 就是这么做的。下面是一个例子，如果我们请求 /foo 这个 URI，请求的内容是 foo，那就返回一个 bar，我们还要把这个应答的状态码设置成 200。

```
server
    .request(and(by("foo"), by(uri("/foo"))))
    .response(and(with(text("bar")), status(200)));
```

在这里，传给 request 和 response 的就不再是一个简简单单的文本，而是一个元素的组合。

所以，传给 request 的，我称之为 RequestMatcher，也就是对请求进行匹配，匹配成功则返回 true，反之返回 false。而传给 response 的，我称之为 ResponseHandler，也就是对应答进行处理，在这里面设置应答中的各种元素。

这就是 Moco 最核心的两个模型。从 Moco 的第一个版本形成开始，一直没有变过。

```
interface RequestMatcher {
    boolean match(Request request);
}

interface ResponseHandler {
    void writeToResponse(Response response);
}
```

从这段代码上，你还可以看到用来组合各个元素的and。学过前面函数式编程的内容，想必你也知道了该如何实现它。除了 and，我还提供了 or 和 not 这样的元素，方便你更好地进行表达。

## 扩展设计

有了基础设计之后，其实 Moco 已经是一个可用的程序库了。从理论上来说，它已经能够完成HTTP 模拟服务器所有的需求了。事实上，当我拿出了 Moco 的第一个版本，就有同事在实际的项目中用了起来。

如同所有开源项目一样，只要有人用，就会有人给出反馈，你就需要去解决它。Moco 就这样，不经意间开启了自己的生命周期。

我在开篇词就说过，软件设计是一门关注长期变化的学问。长期意味着会有需求源源不断地扑面而来。每当有新问题的到来，软件就要去应对这个新的变化，这也是考验软件设计的时候。

第一个变化就是，有人提出要有一个外部的配置文件。Moco 所要做的调整，就是增加一个配置文件，然后要在配置文件和核心模型之间做一个映射。这个变化其实在核心模型上没有任何的改变。学了前面的课程，你也知道，这就相当于给 Moco 增加了一种外部 DSL，只不过，这个 DSL 的语法我采用了 JSON。

正是因为 JSON 配置文件的出现，Moco 有了一个全新的用法，就是把 Moco 当作了一个独立的模拟服务器。后来的很多人其实更熟悉的反而是这种用法，而把 Moco 用在单元测试的这种场景比例就要低一些。也是因为这个独立模拟服务器的用法，Moco 也不再局限于 Java，不同的程序设计语言编写的应用都可以与之进行交互，Moco 的使用范围得到了扩展。

随后，还有人提出了更多功能性上的需求，让 Moco 的能力也得到了极大的提升：

- 有些被模拟的服务不稳定，Moco 支持了一个 proxy 功能，将请求转发给被模拟服务。如果这个服务失效了，就使用本地缓存的信息；
- 有些应答里的字段是根据请求的内容来的，Moco 支持了 template 功能，让使用者自己决定怎样使用哪个信息；

- 有时还要对请求的内容，进行各种匹配。比如，URI 在同一个根目录下，就进行一样的处理，Moco 支持了 match 功能，让使用者自己可以写正则表达式，对请求进行匹配；
- 有人为了方便管理，希望把所有的应答内容放到一个目录下，Moco 支持了 mount 功能，把一个目录挂载在一个 URI ；
- 现在的 REST 开发是主流，Moco 支持了 REST 能力，能够定义资源，更方便地将同一资源的内容定义在一起；
- .....

所有这些内容都是在基础的模型上扩展出来的，基本上都不需要去改动基础模型。不过，有一个功能的拓展影响了基础模型，就是 template。因为它需要根据请求的内容来决定应答的内容，这让原本各自独立的 request 和 response 开始有了关联。

为了适应 template 的需求，我在 ResponseHandler 的接口上增加了 Request，把请求信息带了进来：

```
class SessionContext {  
    private final Request request;  
    private final Response response;  
    ...  
}  
  
interface ResponseHandler {  
    void writeToResponse(SessionContext context);  
}
```

也是由于这个调整，让 Moco 后来有了可以支持录制回放的能力：

```
server  
    .request(by(uri("/record")))  
    .response(record(group("foo")));  
  
server  
    .request(by(uri("/replay")))  
    .response(replay(group("foo")));
```

在这个设置中，我们发给 /record 这个地址的内容就可以记录下来，然后，访问 /replay 这个地址的时候，我们就可以得到刚才记录的内容。由此，Moco 由原来只提供静态设置的模拟服务器，变成了一个能够动态配置的模拟服务器，能力得到了进一步提升。

至此，你已经看到了 Moco 是怎么一点一点长大的。与 2012 年刚刚起步时相比，今天的 Moco 的能力已经强大了许多，但它的内核依然很小，代码量也不大。如果你希望研究一个有设计的代码，不妨从 Moco 入手，这个专栏讲到的不少内容都可以在 Moco 中看到影子。

**Moco 就是根据请求给出应答，只要理解了这么一个简单的逻辑，你就完全可以理解 Moco 在做的事情**，其他的東西都是在这个基础上生长出来的。

## 总结时刻

今天，我给你讲了 Moco 的设计过程。一个好的软件也好，程序库也罢，都是从实际的问题出发的。阻碍一个程序员写出好的程序库的原因，往往是没有找到一个好问题去解决。**程序员不能只当一个问题的解决者，还应该经常抬头看路，做一个问题的发现者。**

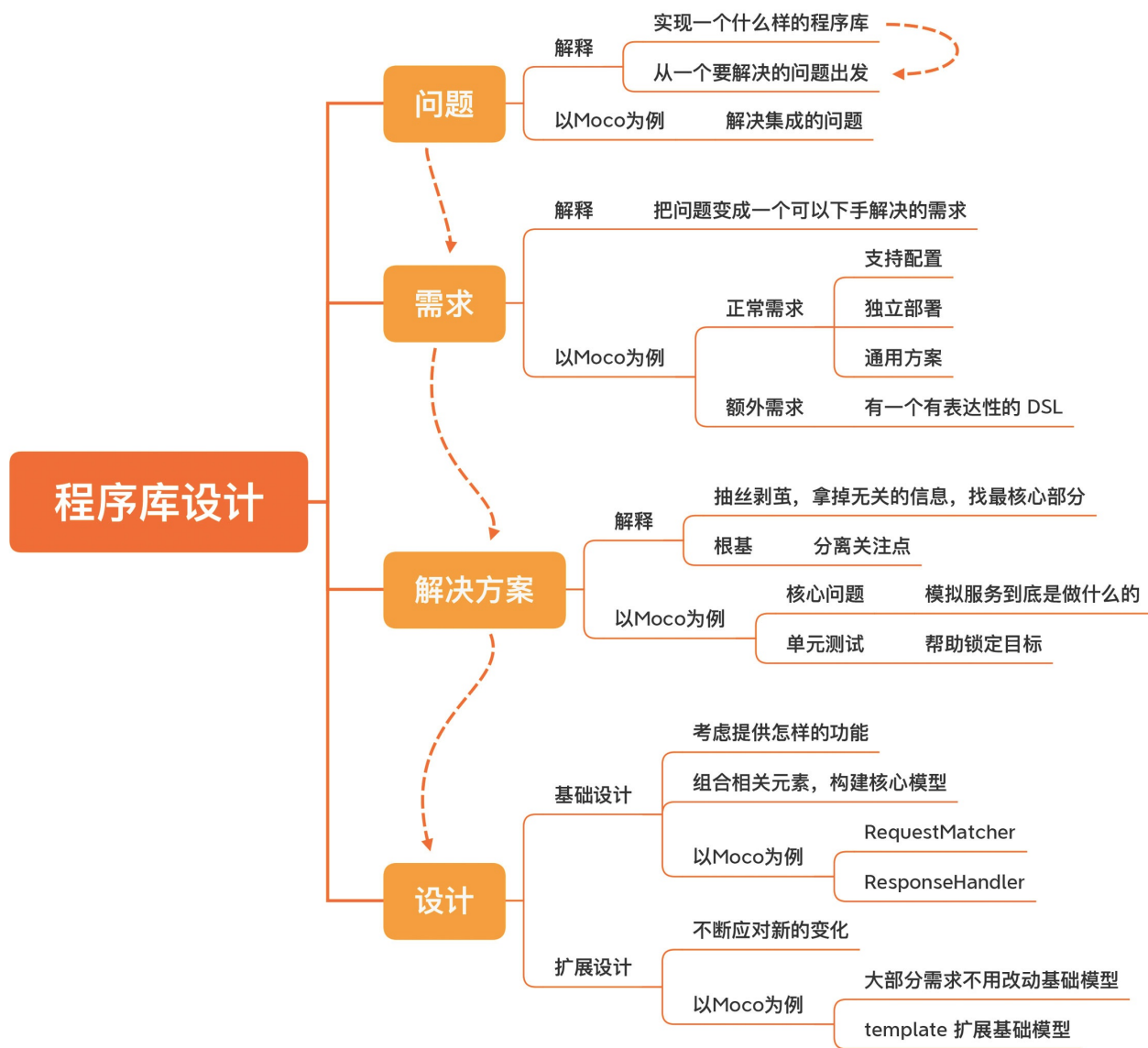
有了问题之后，**需要把问题拆解成可以下手解决的需求**，让自己有一个更明确的目标。然后，我们才是根据这个需求找到一个适当的解决方案。**一个通用的解决方案需要不断地抽丝剥茧，抛开无关的部分，找到核心的部分**，这同样根植于分离关注点。

如果最后的解决方案是一个程序库，那么，我们用测试把程序库要表达的内容写出来，就是最直接的。有了测试，就锁定了目标，剩下的就是让测试通过。

一个好的设计，应该找到一个最小的核心模型，所有其他的内容都是在这个核心模型上生长出来的，越小的模型越容易理解，相对地，也越容易保持稳定。

这一讲，我讲了一个程序库的设计。下一讲，我们再来看看如何设计一个应用。

如果今天的内容你只能记住一件事，那请记住：**注意发现身边的小问题，用一个程序库或工具解决它。**



## 思考题

最后，我想请你抬头看一下路，看看你在开发的过程中，发现过哪些阻碍研发过程的问题呢？欢迎在留言区分享你的想法。

感谢阅读，如果你觉得这一讲的内容对你有帮助的话，也欢迎把它分享给你的朋友。

## 精选留言：

● 人间四月天 2020-08-05 12:45:07

真的很精辟，开发工作是很讲究套路的，从问题，需求，方案，设计，发现问题很关键，太多开发，眼睛里看不到问题，重复开发，功能不复用，不扩展，性能差，开发效率慢，系统质量低，工作中有太多的痛点，痛点即是问题，不追求问题本质，不勤于思考的开发，就是推代码，能跑就行，不管后续维护。如果发现不了问题，更谈不上解决问题，解决方案和设计，就是解决问题，需要积累经验，不断学习，实践，提升解决问题的能力，只有把发现问题和解决问题都做好的开发，才能成为架构师或者leader，更上一层楼。 [2赞]

作者回复2020-08-05 16:22:20

总结得很好！

- 业余爱好者 2020-08-05 07:35:06

记得不错的话，spring mvc test里面也有相似的概念，如RequestMatcher，ResponseHandler,今天才明白原来这是一种函数式编程的dsl。moco已clone,学习一下 [1赞]

作者回复2020-08-05 09:39:58

RequestMatcher和ResponseHandler是模型，函数式的DSL是接口。

- jg4igianshu 2020-08-05 07:17:15

<https://github.com/dreamhead/moco> [1赞]

- 阳仔 2020-08-05 20:25:17

作为程序猿学习能力应该是自带属性，实际工作中，从解决问题出发，锻炼自身的软件设计和开发能力，这是一个层次。

把问题抽象出来提供一个通用的解决方案，并提供程序库出来，这又是一个层次。

自己和自己维护的代码一起进化，这应该是每一个开发者所追求的